List of Abbreviations and Symbols

$A[1..n]$	An array indexed from 1 to n of n elements.
$\mathbb{N}$	Set of all natural numbers, i.e., $\{1, 2, 3, \dots\}$.
$\mathbb{R}$	Set of all real numbers.
$\mathbb{Z}$	Set of all integers, i.e., $\{\dots, -2, -1, 0, 1, 2, \dots\}$.

Modifiers

To help you with what problems to try, problems marked with **[K]** are key questions that tests you on the core concepts, please do them first. Problems marked with **[H]** are harder problems that we recommend you to try after you complete all other questions (or perhaps you prefer a challenge). Good luck!!!

Contents

§1 Introductory problems

Exercise 1.1. [K] You are given an array $A[1..2n-1]$ of integers. Design an algorithm which finds all of the n possible sums of n consecutive elements of A **which runs in time O(n)**. Thus, you have to find the values of all of the sums

$$\begin{aligned} S[1] &= A[1] + A[2] + \ldots + A[n-1] + A[n]; \\ S[2] &= A[2] + A[3] + \ldots + A[n] + A[n+1]; \\ &\vdots \qquad\quad \vdots \qquad\quad \vdots \\ S[n] &= A[n] + A[n+1] + \ldots + A[2n-2] + A[2n-1], \end{aligned}$$

and your algorithm **should run in time O(n)**.

Solution. Here we provide two valid approaches:

Approach 1 - Sliding window: We start by computing $S[1]$ in $O(n)$ by simply adding up $A[1], \ldots, A[n]$, this hence will be our initial window (denote with $(1, n)$).To compute the sum of a window 1 shift towards the end of the array, we only need to include the element to the right of the window ($A[n+1]$) and exclude the left-most element of the window ($A[1]$).

$$\ldots A[i-2] + \underbrace{A[i-1] + A[i] + A[i+1] + \cdots + A[i+n-2]}_{S[i-1]} + A[i+n-1] + \ldots$$

$$\ldots A[i-2] + A[i-1] + \underbrace{A[i] + A[i+1] + \cdots + A[i+n-2] + A[i+n-1]}_{S[i]} + \ldots$$

By generalizing this idea via above, we can compute for each $i = 2, \cdots, n$, $S[i] = S[i-1] - A[i-1] + A[n+i-1]$. As we can compute each subsequent $S[i]$ in $O(1)$ time each, the total of n computations gives an $O(n)$ algorithm.

Approach 2 - Cumulative sum: We can first compute the cumulative sum array $C[0..2n-1]$ where $C[i] = A[1] + A[2] + \cdots + A[i]$. By recognizing that $C[0] = 0$, and for $i = 1, \ldots, n$, $C[i] = C[i-1] + A[i]$, by continuously updating a single variable, we can compute all entries of C in $O(n)$ total time. Then we note that for any $i = 1, \ldots, n$, we have

$$C[i+n-1] = \underbrace{A[1] + A[2] + \cdots + A[i-1]}_{C[i-1]} + \underbrace{A[i] + \cdots + A[i+n-1]}_{S[i]}$$

Then $S[i] = C[i+n-1] - C[i-1]$ which is computable in $O(1)$ thus all the S values can be computed in $O(n)$ also.

Exercise 1.2. [K] Given an array $A[1..n]$ which contains all natural numbers between 1 and $n-1$, design an algorithm that runs in $O(n)$ time and returns the duplicate value, using:

(a) $O(n)$ additional space.

(b) $O(1)$ additional space.

Solution.

(a) Create a zero-initialised array B of length n, which will serve as a frequency table. For each index $i = 1 \ldots n$, record an instance of the value $A[i]$ by incrementing $B[A[i]]$. The duplicate will be the only value j for which $B[j] = 2$.

We perform $O(1)$ work for each of n indices of A, so the time complexity is $O(n)$.

(b) This solution requires us to acknowledge a very important fact which Gauss discovered when he was 7 years old, namely

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

Therefore, we simply add up all elements of A (which we denote as S), then all that remains is computing S subtracted by the sum of all numbers from 1 to $n-1$, i.e., $S - n(n-1)/2$. The remaining value will be the duplicate value.

The sum of all elements of A is computed in $O(n)$ time, followed by one subtraction, so this algorithm runs in $O(n)$. Note that this solution is also acceptable for (a).

Exercise 1.3. [K] You are given an array A of n distinct integers.

(a) You have to determine if there exists a number (not necessarily in A) which can be written as a sum of squares of two distinct numbers from A in two different ways (note: $A[m]^2 + A[k]^2$ and $A[k]^2 + A[m]^2$ counts as a single way) and which runs in time $n^2 \log n$ in the **worst case** performance. Note that the brute force algorithm would examine all quadruples of elements in A and there are $\binom{n}{4} = O(n^4)$ such quadruples.

(b) Solve the same problem but with an algorithm which runs in the **expected time** of $O(n^2)$.

Solution.

(a) There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of distinct indices $1 \leq k < m \leq n$. For each such pair, compute the sum $A[k]^2 + A[m]^2$ and store it in an array of size $n(n-1)/2$. Sort the array using MergeSort. Finally, iterate over the sorted array to determine if any number appears in it at least twice (such occurrences would be consecutive).

There are $O(n^2)$ pairs of indices, so computing them and storing them in an array takes $O(n^2)$. Sorting takes $O(n^2 \log n^2)$, which can be simplified to $O(n^2 \log n)$. The final search takes $O(n^2)$. Therefore the overall time complexity is $O(n^2 \log n)$.

(b) As above, compute each sum $A[k]^2 + A[m]^2$, but instead store the sums in a hash table. Before adding a sum to the hash table, we check whether the same sum already appears there, if so, report `yes`.

For each of $O(n^2)$ pairs of indices, we perform expected $O(1)$ work, so the overall time complexity is expected $O(n^2)$.

Exercise 1.4. [H] You are conducting an election among a class of n students. Each student casts precisely one vote by writing their name, and that of their chosen classmate on a single piece of paper. We assume that students are not allowed to vote for themselves. However, the students have forgotten to specify the order of names on each piece of paper – for instance, "Alice Bob" could mean that Alice voted for Bob, or that Bob voted for Alice!

(a) Show how you can still uniquely determine how many votes each student received.

(b) Hence, explain how you can determine which students did not receive any votes. Can you determine who these students voted for?

(c) Suppose every student received at least one vote. What is the maximum possible number of votes received by any student? Justify your answer.

(d) Using parts (a) and (c), or otherwise, design an algorithm that constructs a list of votes of the form "X voted for Y" consistent with the pieces of paper. Specifically, each piece of paper should match up with precisely one of these votes. If multiple such lists exist, produce any. An $O(n^2)$ algorithm earns partial credit, but you should aim for an $O(n)$ algorithm.

Solution

(a) If a student's name appears on x pieces of paper, then the student received $x - 1$ votes since each student voted precisely once.

(b) If a student did not receive any votes, their name only appears on precisely one piece of paper. The name of the other student is who they voted for.

(c) If every student received at least one vote, then at least n distinct pieces of paper are required to correspond to these votes. There are no more pieces of paper to be distributed, so every student received exactly one vote. Hence, each student also received a maximum of **one vote**.

(d) Suppose every student received at least one vote. Then, by (c), every student received exactly one vote. By considering the votes as an undirected graph (where each student is a vertex and every vote is an edge between two students), or otherwise, we can see that every student appears on precisely two pieces of paper. This corresponds to a set of disjoint cycle graphs where students are vertices and pieces of paper are edges between students.

Pick any student s appearing on two pieces of paper, and arbitrarily choose one of their pieces of paper as their vote. Suppose they voted for t. We are now left with a single choice for t's vote. We can repeatedly follow these pieces of paper until we arrive back to s. We then repeat with another student appearing on two pieces of paper until all votes have been resolved. We can do this in $O(n)$ altogether, for instance using a (simplified) Depth-First Search (DFS).

Now we combine this with part (b) to obtain an algorithm for the general case. We repeatedly check if a student has no votes (by counting votes) and resolve their vote. Once we reach a point where this is no longer possible, we know every student received at least one vote, and use the algorithm above.

This can be done in $O(n^2)$ by repeatedly taking $O(n)$ to identify a student who has no votes, or more cleverly in $O(n)$ as follows. We keep a count, for each student, how many pieces of paper they appear on and maintain a queue of students who appear on only one piece of paper. We can initially populate this queue in $O(n)$. Then, we repeatedly process the front student of the queue by removing their vote. Note that this *only changes the vote count of the person they voted for*, so we simply decrease their count. If their count reaches 1, we push them onto the queue. Hence, we process each student, updating counts and the queue in $O(1)$ so this step is $O(n)$ as well, giving an $O(n)$ algorithm.

> **Hint** — *first, use part (c) to consider how you would solve it in the case where every student received at least one vote. Then, apply part (b).*

Exercise 1.5. [H] You are at a party attended by n people (not including yourself), and you suspect that there might be a celebrity present. A *celebrity* is someone known by everyone, but who does not know anyone else present. Your task is to work out if there is a celebrity present, and if so, which of the n people present is a celebrity. To do so, you can ask a person X if they know another person Y (where you choose X and Y when asking the question).

(a) Show that your task can always be accomplished by asking no more than $3n - 3$ such questions, even in the worst case.

(b) Show that your task can always be accomplished by asking no more than $3n - \lfloor \log_2 n \rfloor - 3$ such questions, even in the worst case.

Solution. Assume that the people are assigned a number from 1 to n.

(a) We make the following observation.

Observation. *There can be at most one celebrity.*

Suppose that there are two celebrities, say A and B. Since A is a celebrity, B must know A since everyone knows the celebrity. But then this implies that A cannot be a celebrity since a celebrity does not know anyone. So there can only be *at most* one celebrity. With this observation, we proceed as follows.

Arbitrarily pick any two people, say A and B. Ask if A knows B.

- If A knows B, then we know that A cannot be a celebrity.
- If A does not know B, then we know that B cannot be a celebrity.

In either case, we can eliminate one person from our *celebrity candidate* list. Since there are n people, we can ask $n-1$ people to find the *celebrity candidate*. Let this candidate be C.

We now check if C is indeed the celebrity (it could very well be the case that C is not the celebrity and as such, no celebrity exists). For each person X in the party, we ask the following question:

Does X know C?

- If X does not know C, then we conclude that C cannot be the celebrity and thus, no celebrity is present.
- Otherwise, C may still be the celebrity.

We finally need to check if C knows anyone in the party. Again, choosing every person X in the party, we ask the following question:

Does C know X?

- If C knows X, then we conclude that C cannot be the celebrity and thus, no celebrity is present.
- If C does not know X, we continue until we have exhausted everyone in the party.

We only conclude that C must be the only celebrity once we have concluded that C does not know anyone. In the worst case, we consume $n-1$ questions to determine who the potential celebrity candidate is and then $2(n-1)$ questions to verify whether C is indeed the celebrity. Thus, in the worst case, we use $(n-1)+2(n-1)=3n-3$ questions in total.

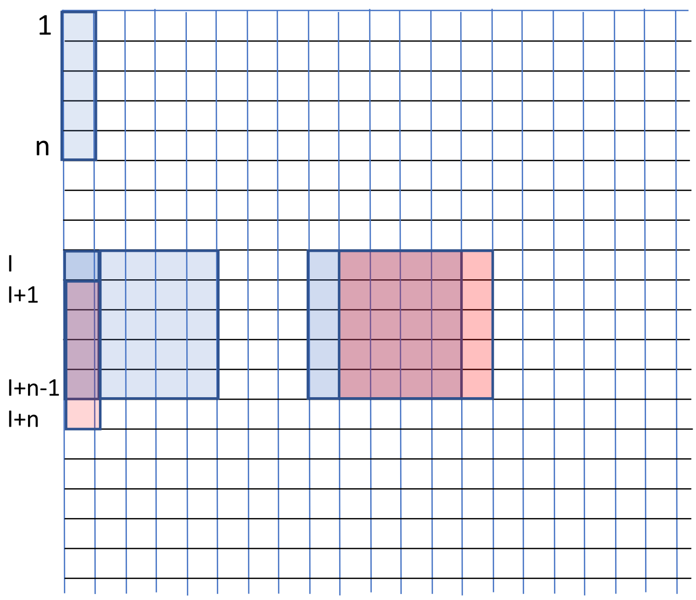
(b) We arrange n people present as leaves of a **balanced full tree**, i.e., a tree in which every node has either 2 or 0 children and the depth of the tree is as small as possible. To do that compute $m=\lfloor \log_2 n \rfloor$ and construct a perfect binary three with $2^m \leq n$ leaves. If $2^m < n$ add two children to each of the leftmost $n-2^m$ leaves of such a perfect binary tree. In this way you obtain $2(n-2^m)+(2^m-(n-2^m)) = 2n-2^{m+1}+2^m-n+2^m = n$ leaves exactly, but each leaf now has its pair, and the depth of each leaf is $\lfloor \log_2 n \rfloor$ or $\lfloor \log_2 n \rfloor + 1$. For each pair we ask if, say, the left child knows the right child and, depending on the answer as in (a) we promote the potential celebrity one level closer to the root. It will again take $n-1$ questions to determine a potential celebrity, but during the verification step we can save $\lfloor \log_2 n \rfloor$ questions (one on each level) because we can reuse answers obtained along the path that the potential celebrity traversed through the tree. Thus, $3n-3-\lfloor \log_2 n \rfloor$ questions suffice.

Exercise 1.6. [H] There are N teams in the local cricket competition and you happen to have N friends that keenly follow it. Each friend supports some subset (possibly all, or none) of the N teams. Not being the sporty type – but wanting to fit in nonetheless – you must decide for yourself some subset of teams (possibly all, or none) to support. You don't want to be branded a copycat, so your subset must not be identical to anyone else's. The trouble is, you don't know which friends support which teams, so you can ask your friends some questions of the form "Does friend A support team B?" (you choose A and B before asking each question). Design an algorithm that determines a suitable subset of teams for you to support and asks as few questions as possible in doing so.

Solution. Suppose your friends are numbered 1 to N and the teams are also numbered 1 to N. Then, for each i, ask friend i if they support team i. If they do, we choose not to support them and if they don't, we do support them. Clearly, this subset of teams is different to all of our friends', and it uses N queries, which is the minimal possible for any deterministic solution (we must have some information about each friend). Note: this method is called "diagonalisation".

Exercise 1.7. [H] You are in a square orchard of $4n$ by $4n$ equally spaced trees. You want to purchase apples from precisely n^2 many of those trees, which also form a square. Fortunately, the owner is allowing you to choose such a square anywhere in the orchard and you have a map with the number of apples on each tree. Your task is to choose a square that contains the largest amount of apples and which runs in time $O(n^2)$. Note that the brute force algorithm would run in time $\Theta(n^4)$.

Solution. We start by noting that there is a heavy overlap between such possible squares and we should use this fact to compute the number of apples in all of such squares in an efficient way. Consider to the figure below.



Setup: Let us visualize the orchard as a $4n \times 4n$ discrete grid and let $C(i,j)$ denote the cell at the coordinate (i,j). Also let $A[i,j]$ denote the amount of apples at $C[i,j]$.

Step 1: Now, we start by examining the first column by computing the sum $\alpha(1,1) = \sum_{k=1}^{n} A[k,1]$, (corresponding to cells $C[1,1]$ to $C[n,1]$ shown in blue in the top left corner of the orchard map) which takes $n-1 = O(n)$ additions. We then compute the number of apples $\alpha(i,1)$ in all rectangles $r(i,1)$ consisting of cells $C[i,1]$ to $C[i+n-1,1]$ for all i such that $2 \le i \le 3n+1$, starting from $\alpha(1,1)$ and using recurrence

$$\alpha(i+1,1) = \alpha(i,1) - A[i,1] + A[i+n,1].$$

We know that the recurrence is valid as the rectangle (denoted with $r(i,1)$) consisting of cells $C[i,1]$ to $C[i+n-1,1]$ and rectangle $r(i+1,1)$ consisting of cells $C[i+1,1]$ to $C[i+n,1]$ in the first column overlap and differ only in the first square of $r(i)$ and the last square of $r(i+1)$(this idea is similar to Question 1.1

but embedded 2 dimensions). Since each recursion step involves only one addition and one subtraction, this can all be done in $O(n)$ steps.

Step 2: We can now apply the same procedure to each different column j, thus obtaining the number of apples $\alpha(i,j)$ in every rectangle consisting of cells $C(i,j)$ to $C(i+n-1,j)$. As each column requires $O(n)$ many computations, it takes $O(n^2)$ many computations in total.

Step 3: Now for $1 \leq i \leq 3n+1$, we compute $\beta(i,1) = \sum_{p=1}^{n} \alpha(i,p)$. This gives the total number of apples acquired in the square with corner vertices at cells $C(i,1), C(i,n), C(i+n-1,1)$ and $C(i+n-1,n)$ (square with vertices along a single column). For n such computations it takes $O(n)$ additions each, thus the total number of additions then runs in $O(n^2)$.

Step 4: So for each fixed i such that$1 \leq i \leq 3n+1$, with the value of $\beta(i,1)$, we can now compute $\beta(i,j)$ for all $2 \leq j \leq 3n+1$ using recursion

$$\beta(i,j+1) = \beta(i,j) - \alpha(i,j) + \alpha(i,j+n).$$

because the two adjacent squares overlap, except for the first column of the first square and the last column of the second square (overlap between red and blue square), the rest of the considered squares overlap. Each step of recursion takes only one addition and one subtraction so the whole recursion takes $O(n)$ many steps. Thus, recursions for all $1 \leq i \leq 3n+1$ takes $O(n^2)$ many operations. (idea is again similar to Question 1.1 but in 2 dimensions).

Step 5: Lastly,we only require to find the largest value of $\beta(i,j)$ among all $1 \leq i,j \leq 3n+1$, giving a complexity of $O((3n)^2) = O(n^2)$ for finding the maximum.

As all required steps of the algorithm runs in a complexity of $O(n^2)$, the total complexity of the algorithm is then $O(n^2)$ as required.

§2 Searching algorithms

Exercise 2.1. [K] Let M be an $n \times n$ matrix of distinct integers $M(i,j)$, $1 \leq i \leq n$, $1 \leq j \leq n$. Each row and each column of the matrix is sorted in the increasing order, so that for each row i, $1 \leq i \leq n$,

$$M(i,1) < M(i,2) < \ldots < M(i,n)$$

and for each column j, $1 \leq j \leq n$,

$$M(1,j) < M(2,j) < \ldots < M(n,j).$$

You need to determine whether M contains an integer x in $O(n)$ time.

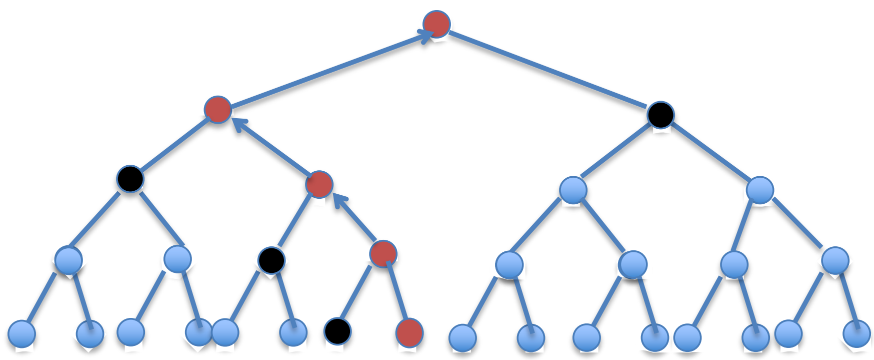
Solution. Consider $M(1,n)$ (i.e., top right cell).

- If $M(1,n) = x$, we are done.
- If $M(1,n) < x$, the number x is not found in the top row because $M(1,1) < M(1,2) < \ldots < M(1,n) < x$. We can therefore ignore this row.
- If $M(1,n) > x$, then similarly x cannot be found in the rightmost column because all other elements there are larger than $M(1,n)$. Thus the last column can be ignored.

In the worst case, the sum of the width and height of the search table is reduced by one. We continue in this manner until either x is found or we reach an empty table and thus ascertain that x does not occur in the table. Since the initial sum of the height and the width of the table is $2n$ and at each step we make only one comparison, the algorithm takes $O(n)$ many steps.

Exercise 2.2. [K] You are given an array of $A[1..2^n]$ distinct integers. You are tasked to find the largest and the second largest number using only $2^n + n - 2$ comparisons.

Solution. Consider the figure below.



We see a complete binary tree with 2^n leaves and $2^n - 1$ internal nodes and of depth n (the root has depth 0). We then place all the numbers at the leaves, compare each pair and "promote" the larger element (shown in red) to the next level and proceed in such a way till you reach the root of the tree, which will contain the largest element (like a sports tournament!!).

Clearly, each internal node is a result of one comparison and there are $2^n - 1$ many nodes thus also the same number of comparisons so far. Now just note that the second largest element must be among the black nodes which were compared with the largest element along the way - all elements underneath them must be smaller or equal to the elements shown in black. There are n many such elements so finding the largest among them will take $n - 1$ comparisons by brute force.

In total requires exactly $2^n + n - 2$ many comparisons.

Exercise 2.3. **[K]** Assume you have an array of $2n$ distinct integers. Find the largest and the smallest number using $3n-2$ comparisons only.

Solution. Consider first the brute force algorithm:

- Compare the first two elements $A[1]$ and $A[2]$, and set m as the smaller of them and M as the larger.
- For each of the following $2n-2$ elements, compare it with both m and M, updating either if necessary.

In the best case, we will update m at each step, making it unnecessary to ever compare against M, and resulting in a total of $2n-1$ comparisons. However, this algorithm is not acceptable because in the worst case, each of $2n-2$ elements requires two comparisons, for a total of $2(2n-2)+1 = 4n-3$ comparisons.

Instead, we first form n pairs and compare the two elements of each pair, putting the smaller into a new array S and the larger into a new array L. Note that the smallest of all $2n$ elements must be in S and the largest in L, and we have made n comparisons. We now use two linear searches for the minimum of S and the maximum of L, each taking $n-1$ comparisons. In total this takes $n+n-1+n-1 = 3n-2$ comparisons.

Exercise 2.4. **[K]** Let $f : \mathbb{N} \to \mathbb{Z}$ be a monotonically increasing function. That is, for all $i \in \mathbb{N}$, we have that $f(i) < f(i+1)$. Our goal is to find the smallest value of $i \in \mathbb{N}$ so that $f(i) \geq 0$. Design an $O(\log n)$ algorithm to find the value of i so that $f(i) \geq 0$ for the first time.

> **Hint —** *If the function notation is confusing, it might help to consider this problem in the context of an* ***array****. Specifically, how are arrays and functions closely related?*

Solution. One possible strategy is to directly compute $f(1), f(2), f(3), \ldots$. This works but takes $O(n)$ time in the worst case. So we need to be slightly smarter. Observe that, since f is monotonically increasing, then we can compute the "middle" value (whatever that means) and see which side we need to search on. This is because, if $f(k) > 0$, then $f(m) > 0$ for all $m > k$. So these elements cannot be the first time that the function changes sign. If $f(k) < 0$, then $f(n) < 0$ for all $n < k$. So these elements cannot be the first time that the function changes sign. This is the basis for our binary search.

The only issue is that we don't really have a notion of a "middle" value since the function is unbounded. To account for this, we can find an artificial `high` value by repeated doubling until $f(\texttt{high})$ becomes positive. That is, we compute the following values:

$$f(1),\ f(2),\ f(4),\ f(8), \ldots, f(\texttt{high}).$$

Therefore, we know that the index must occur between `high`/2 and `high`. We repeat this process until we hit one element, which is the index that we are after. Computing `high` takes $O(\log n)$ many computations and in each subroutine, we are making $O(\log n)$ many computations. So, the entire algorithm takes $O(\log n)$ time.

Exercise 2.5. **[K]** You're given an array $A[1..n]$ of integers, and you are required to answer a series of n queries, each of the form: "how many elements of the array have a value between L and R (inclusively)?", where L and R are integers. Design an $O(n \log n)$ algorithm that answers all n of these queries.

Solution. We start by sorting A in $O(n \log n)$ in ascending order, using Merge Sort. Then, for each query, we can 2 binary searches to find the indexes (denote with (i, j)) of the:

- First element with value **no less** than L; and
- First element with value **strictly greater** than R.

The difference between these indices is the answer to the query, i.e., $j - i$. Note that if your binary search hits L you have to see if the preceding element is smaller than L; if it is also equal to L, you have to continue the binary search (going towards the smaller elements) until you find the first element equal to L. A similar observation applies if your binary search hits R.

Each binary search then takes $O(\log n)$ so for n queries in total, the algorithm runs in $O(n \log n)$ overall.

Exercise 2.6. [H] Assume you are given two arrays A and B, each containing n distinct positive numbers and the equation $x^8 - x^4y^4 = y^6 + x^2y^2 + 10$. Design an algorithm which runs in time $O(n \log n)$ which finds if A contains a value for x and B contains a value for y that satisfy the equation.

Solution. Let

$$f(x, y) = y^6 + x^4y^4 + x^2y^2 + 10 - x^8,$$

so the original equation is $f(x, y) = 0$. The crucial observation is that for a fixed value $x = x_0$, $f(x_0, y)$ is an increasing function of y. Our algorithm is then:

(a) Sort array B using MergeSort.

(b) For each index $i = 1..n$ of A, we fix $x = A[i]$ and perform a binary search on B.

- Let index m be the midpoint of the current subarray $B[l..r]$ (initially $B[1..n]$).
- If $f(A[i], B[m]) = 0$, we have found a solution.
- If $f(A[i], B[m]) < 0$, the value at the midpoint is too small, and all earlier entries of B will also be too small. Therefore, we recurse on $B[(m + 1)..r]$.
- Conversely, if $f(A[i], B[m]) > 0$, the value at the midpoint is too large, and all later entries of B are also too large, so we recurse on $B[l..(m - 1)]$.

We perform n binary searches, each taking $O(\log n)$ time for a total of $O(n \log n)$. Note that the evaluation of $f(A[i], B[m])$ always takes constant time.

Exercise 2.7. [H] You are a fisherman, trying to catch fish with a net that is W meters wide. Using your advanced technology, you know that the positions of all N fish in the sea can be represented as integers on a number line. There may be more than one fish at the same location.
To catch the fish, you will cast your net at position x, and will catch all fish with positions between x and $x + W$, **inclusive**. Given N, W and an array $X[1..N]$ denoting the positions of fish in the sea, give an $O(N \log N)$ algorithm to find the maximum number of fish you can catch by casting your net once.
For example, if $N = 7, W = 3$ and $X = [1, 11, 4, 10, 6, 7, 7]$, then the most fish you can catch is 4: by placing your net at $x = 4$, you will catch one fish at position 4, one fish at position 6 and two fish at position 7.

Solution. First, sort array X using MergeSort in $O(N \log N)$ time. We know that it is optimal to cast our net starting from the same position as a fish. We can use a "two pointers" approach: we keep variables l and r, which designate the leftmost and rightmost fish in our net, at positions $X[l]$ and $X[r]$ respectively.

Initially $l = 1$, and we iterate forward to find the largest r such that the rth fish in the sorted array X is still within our net, that is, the largest r such that $X[r] \leq 1 + W$. We then repeatedly increment l, and respectively repeatedly increment r to include all fish that fit within the net starting at the lth fish. At each stage, we know we can catch $r - l + 1$ fish, so we take the maximum among these.

The initial search for r takes $O(N)$ time. For each subsequent pair of values (l, r), we perform only a constant amount of work. Since l and r are only ever incremented, and they are each incremented at most

N times, the rest of the algorithm is also $O(N)$, so the overall time complexity is $O(N)$.

Exercise 2.8. [H] Your army consists of a line of N giants, each with a certain height. You must designate precisely $L \leq N$ of them to be leaders. Leaders must be spaced out across the line; specifically, every pair of leaders must have at least $K \geq 0$ giants standing in between them. Given N, L, K and the heights $H[1..N]$ of the giants in the order that they stand in the line as input, find the *maximum* height of the *shortest* leader among all valid choices of L leaders. We call this the *optimisation* version of the problem.

For instance, suppose $N = 10, L = 3, K = 2$ and $H = [1, 10, 4, 2, 3, 7, 12, 8, 7, 2]$. Then among the 10 giants, you must choose 3 leaders so that each pair of leaders has at least 2 giants standing in between them. The best choice of leaders has heights 10, 7 and 7, with the shortest leader having height 7. This is the best possible for this case.

(a) In the *decision* version of this problem, we are given an additional integer T as input. Our task is to decide if there exists some valid choice of leaders satisfying the constraints whose shortest leader has height no less than T.
Give an algorithm that solves the decision version of this problem in $O(N)$ time.

(b) Hence, show that you can solve the optimisation version of this problem in $O(N \log N)$ time.

Solution.

(a) Notice that for the decision variant, we only care for each giant whether its height is at least T, or less than T: the actual value doesn't matter. Call a giant *eligible* if their height as at least T.

We sweep from left to right, taking the first eligible giant we can, then skipping the next K giants and repeating. We return `true` if the total number of giants we obtain from this process is at least L, or `false` otherwise. Each giant is processed in constant time, so the algorithm is clearly $O(N)$.

(b) Observe that the optimisation problem corresponds to finding the largest value of T for which the answer to the decision problem is `true`.

Suppose our decision algorithm returns `true` for some T. Then clearly it will return true for all smaller values of T as well: since every giant that is eligible for this T will also be eligible for smaller T. Hence, we can say that our decision problem is *monotonic* in T.

Thus, we can use binary search to work out the maximum value of T where our decision problem returns `true`. Note that it suffices to check only heights of giants as candidate answers: the answer won't change between them. Thus, we can sort our heights in $O(N \log N)$ and binary search over these values, deciding whether to go higher or lower based on a run of our decision problem. Since there are $O(\log N)$ iterations in the binary search, each taking $O(N)$ to resolve, our algorithm is $O(N \log N)$ overall.

Exercise 2.9. [H] Let A be an array of n integers, not necessarily distinct or positive. Design a $\Theta(n \log n)$ algorithm that returns the maximum sum found in any contiguous subarray of A.

Note that there is an $O(n)$ algorithm that solves this problem; however, the technique used in that solution is much more involved and will be taught in due time.

Solution. Our apologies, this question was included in Tutorial 1 in error. It will be moved to Tutorial 2, as the required techniques will only be taught in Lecture 3.

§3 Sorting algorithms

Exercise 3.1. [K] You are given an array $A[1..n]$ of integers and another integer x.

(a) Design an $O(n \log n)$ algorithm (in the sense of the worst case performance) that determines whether or not there exist two elements in A whose sum is exactly x.

(b) Design an algorithm that accomplishes the same task, but runs in $O(n)$ **expected** (i.e., average) time.

Solution.

(a) Note that a brute force solution considers all possible pairs of $(A[i], A[j])$ does not suffice as it runs in $O(n^2)$ time. Instead, we start by sorting the array in ascending order, which can be done in $O(n \log n)$ in the worst case using an algorithm such as Merge Sort. Here we give 2 valid approaches.

Approach 1: For each element a in the array, we can check if there exists an element $x - a$ also in the array in $O(\log n)$ time using binary search. The only special case is if $a = x - a$ (i.e. $x = 2a$), where we just need to check the two elements adjacent to a in the sorted array to see if another a exists.

Hence, we take at most $O(\log n)$ time for each element, so this part is also $O(n \log n)$ time in the worst case, giving an $O(n \log n)$ algorithm.

Approach 2: Alternatively, we add the smallest and the largest elements of the array. If the sum exceeds x no solution can exist involving the largest element; if the sum is smaller than x then no solution can exist involving the smallest element. Thus, if this sum is not equal to x we can eliminate 1 element.

After at most $n - 1$ many such steps you will either find a solution or will eliminate all elements except one, thus verifying no such elements exist. This takes $O(n)$ time in the worst case, so the overall runtime is $O(n \log n)$ as the sorting dominates.

(b) We take a similar approach in (a), except we use a hash map (or hash table, denoted H) to check if elements exist in the array (rather than sorting it): each insertion and lookup takes $O(1)$ **expected** time. We again provide 2 valid approaches.

Approach 1: At index $i \in \{1, 2, \dots, n\}$, we assume the previous $i - 1$ elements of A are already stored in H. Then we check if $x - A[i]$ is in H in expected $O(1)$, then insert $A[i]$ into H, also in expected $O(1)$.

As this process will take n insertions and n look-ups in the worst case, we conclude that the algorithm runs in $O(2n) = O(n)$ **expected** time.

Approach 2: Alternatively, we hash all elements of A and store its occurrence frequency. We then go through elements of A again, this time for each element a we check if $x - a$ is in H. However, if $2a = x$, we must also check if at least 2 copies of a appear in the corresponding slot H.

As this process will take again n insertions and n look-ups in the worst case, hence the algorithm runs in $O(n)$ **expected** time.

Exercise 3.2. [K] Given two arrays of n integers, design an algorithm that finds out in $O(n \log n)$ steps if the two arrays have an element in common.

Solution. Sort one of the arrays in $O(n \log n)$ using merge sort. Without the loss of generality, suppose that we sort array A. Then, for each element in array B, perform a binary search on A to locate the element in B. If there is a match, then return YES. If we have exhausted all of the elements in B without a match, return NO. Sorting one of the arrays takes $O(n \log n)$. Then, for each subsequent element, we perform *at most* $\log n$ many comparisons. So searching for a match takes *at most* $n \log n$ many comparisons, which yields an $O(n \log n)$ algorithm.

Exercise 3.3. [K] Suppose that you are taking care of n kids, who took their shoes off. You have to take the kids out and it is your task to make sure that each kid is wearing a pair of shoes of the right size (not necessarily their own, but one of the same size). All you can do is to try to put a pair of shoes on a kid, and see if they fit, or are too large or too small; you are NOT allowed to compare a shoe with another shoe or a foot with another foot. Describe an algorithm whose expected number of shoe trials is $\mathrm{O}(n \log n)$ which properly fits shoes on every kid.

Hint — *Try "double"* QUICKSORT*; one which uses a foot as a pivot for shoes and another one which uses a shoe as a pivot for feet.*

Solution. This is done by a "double QUICKSORT" as follows. Pick a shoe and use it as a pivot to split the kids into three groups: those for whom the shoe was too large, those who fit the shoe and those for whom the shoe was too small. Then pick a kid for whom the shoe was a fit and let him try all the shoes, splitting them in three groups as well: shoes that are too small, shoes that fit him and the shoes which were too large for him. Continue this process with the first group of kids and first group of shoes and then also the third group of shoes with the third group of kids. If kids and shoes are picked randomly, the expected time complexity will be $O(n \log n)$.

Exercise 3.4. [H] Given n real numbers $x_1, \ldots, x_n$ in the interval $[0, 1)$, devise an algorithm that runs in linear time which outputs a permutation of the n numbers, say $y_1, \ldots, y_n$, such that

$$\sum_{i=2}^{n} |y_i - y_{i-1}| < 2.$$

Hint — *this is easy to do in $O(n \log n)$ time: just sort the sequence in ascending order. In this case, $\sum_{i=2}^{n} |y_i - y_{i-1}| = \sum_{i=2}^{n} (y_i - y_{i-1}) = y_n - y_1 \leq 1 - 0 = 1$. Here $|y_i - y_{i-1}| = y_i - y_{i-1}$ because all the differences are non-negative, and all the terms in the sum except the first and the last one cancel out. To solve this problem, one might think about tweaking the* BUCKETSORT *algorithm, by carefully avoiding sorting numbers in the same bucket.*

Solution. We start by splitting the interval $[0, 1)$ into n equal buckets, namely

$$B = \{b_0, b_1, \ldots, b_{n-1}\} = \left\{ \left[0, \frac{1}{n}\right), \left[\frac{1}{n}, \frac{2}{n}\right), \ldots, \left[\frac{n-1}{n}, 1\right) \right\}$$

Then we consider the function $b : \mathbb{R} \to \{0, \ldots, n-1\}$ which computes $b(x) = k$ such that $x \in b_k$. Then we consider that

Claim — The value x_i belongs to bucket number $b(x_i) = \lfloor n\, x_i \rfloor$.

Proof. From the definition, we know that x_i is in the bucket k if and only if

$$\frac{k}{n} \leq x_i < \frac{k+1}{n} \implies k \leq n\, x_i < k+1 \quad \text{then} \quad k = \lfloor n\, x_i \rfloor.$$

Therefore we can form n pairs $\langle x_i, b(x_i)\rangle$, each in constant time. Then we can now sort these pairs according to their bucket number $b(x_i)$; since all bucket numbers are less than n, CountingSort does that in linear time. One can show that this sequence already satisfies the condition of the problem, but to make things simpler we do another extra step. We go through the sequence and in each bucket we find the smallest and the largest element; this can clearly be done in linear time.

We now slightly change the ordering of each bucket: we always start with the smallest element in that bucket and finish with the largest element (leaving all other elements in the same order). Discard the bucket numbers, leaving only a sequence y_j of real numbers between 0 and 1, which are the original x_i rearranged.

We now prove that this sequence satisfies $\sum_{i=2}^{n} |y_i - y_{i-1}| < 2$. We start by spliting this sum into two parts:

$$\sum_{i=2}^{n} |y_i - y_{i-1}| = \sum_j \left(|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in the same bucket}\right) + \sum_j \left(|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in different buckets}\right).$$

Note that there can be at most $n-1$ pairs of consecutive elements y_i, y_{i-1} which are in the same bucket (when all elements are in the same bucket). Whenever two elements y_i, y_{i-1} are in the same bucket, $|y_i - y_{i-1}|$ is at most equal to the size of the bucket, i.e, $< \frac{1}{n}$.

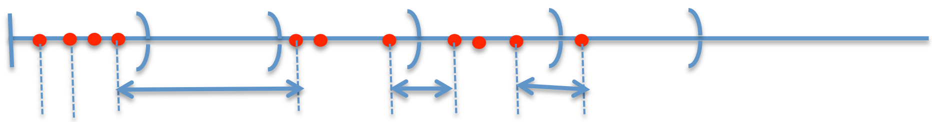
Thus,

$$\sum_j \left(|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in the same bucket}\right) < \frac{n-1}{n} < 1.$$

Also,

$$\sum_j \left(|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in different buckets}\right) \leq 1$$

because each such pair joins the largest entry of one bucket to the smallest entry of the next non-empty bucket; as a result, the intervals in question are disjoint, as shown in the figure below.



Thus the total sum is smaller than $1 + 1 = 2$.

Exercise 3.5. [H] Given n real numbers $x_1, \ldots, x_n$ where each x_i is a real number in the interval $[0, 1)$, devise an algorithm that runs in linear time and that will output a permutation of the n numbers, say $y_1, \ldots, y_n$, such that $\sum_{i=2}^{n} |y_i - y_{i-1}| < 1.01$.

Hint — *Refer to the previous exercise. How do you alter the previous exercise to do this problem?*

Solution. Exactly the same as the previous problem, just instead of using n buckets, use $100n$ many buckets.

§4 Complexity analysis

Exercise 4.1. [K] Determine if $f(n) = \Omega(g(n))$, $f(n) = O(g(n))$, both (i.e. $f(n) = \Theta(g(n))$) or neither for the following pairs. Justify your answers.

$f(n)$	$g(n)$
$(\log_2 n)^2$	$\log_2(n^{\log_2 n}) + 2\log_2 n$
n^{100}	$2^{n/100}$
$\sqrt{n}$	$2^{\sqrt{\log_2 n}}$
$n^{1.001}$	$n \log_2 n$
$n^{(1+\sin(\pi n/2))/2}$	$\sqrt{n}$

Hint — *You might find the following inequality useful: if $f(n), g(n), c > 0$ then $f(n) < c\, g(n)$ if and only if $\log f(n) < \log c + \log g(n)$.*

(a) We see that

$$\begin{aligned} g(n) &= \log_2 n \cdot \log_2 n + 2\log_2 n \\ &= \Theta((\log_2 n)^2) = \Theta(f(n)), \end{aligned}$$

since $2\log_2 n < (\log_2 n)^2$ for sufficiently large n.

(b) We show that $f(n) = O(g(n))$. To do this, we want to find some constants $c, N > 0$ such that, for every $n > N$, $f(n) < c \cdot g(n)$. Since log is a monotonically increasing function, $\log f(n) < \log g(n)$ immediately implies that $f(n) < g(n)$. We see that

$$\begin{aligned} \log_2 f(n) &= \log_2\left(n^{100}\right) \\ &= 100\log_2 n, \\ \log_2 g(n) &= \log_2\left(2^{n/100}\right) \\ &= \frac{n}{100}. \end{aligned}$$

It therefore suffices to show that, for sufficiently large n, $10000\log_2 n < n$. We see that

$$\lim_{n\to\infty} \frac{10000\log_2 n}{n} = \lim_{n\to\infty} \frac{10000}{n\log 2} = 0,$$

by L'Hôpital's rule. In other words, there exist some $N > 1$ such that, for all $n > N$, $10000\log_2 n/n < 1$ and thus, $\log_2 f(n) < \log_2 g(n)$ which implies that $f(n) < g(n)$ and so, $f(n) = O(g(n))$.

(c) We show that $f(n) = \Omega(g(n))$. This amounts to showing that $\sqrt{n} > c \cdot 2^{\sqrt{\log_2 n}}$ for some $c > 0$ and for all sufficiently large n. Since log is monotonically increasing, this is equivalent to showing that

$$\log_2 \sqrt{n} = \frac{1}{2}\log_2 n > \log_2 c + \sqrt{\log_2 n} = \log_2\left(c \cdot 2^{\sqrt{\log_2 n}}\right).$$

Taking $c = 1$, it is clear that $\log_2 n$ grows asymptotically faster than $\sqrt{\log_2 n}$. Hence, $\log_2 \sqrt{n} > \log_2\left(2^{\sqrt{\log_2 n}}\right)$ which implies that $f(n) = \Omega(g(n))$.

(d) We again wish to show that $n^{1.001} = \Omega(n \log n)$, i.e., that $n^{1.001} > cn \log n$ for some c and all sufficiently large n. Since $n > 0$ we can divide both sides by n, so we have to show that $n^{0.001} > c \log n$. We again take $c = 1$ and show that $n^{0.001} > \log n$ for all sufficiently large n, which is equivalent to showing that $\log n / n^{0.001} < 1$ for sufficiently large n. To this end we use the L'Hôpital's to compute the limit

$$\begin{aligned}\lim_{n\to\infty} \frac{\log n}{n^{0.001}} &= \lim_{n\to\infty} \frac{(\log n)'}{(n^{0.001})'} = \lim_{n\to\infty} \frac{\frac{1}{n}}{0.001 n^{0.001-1}} \\ &= \lim_{n\to\infty} \frac{\frac{1}{n}}{0.001\frac{1}{n} \cdot n^{0.001}} = \lim_{n\to\infty} \frac{1}{0.001 \cdot n^{0.001}} = 0.\end{aligned}$$

Since $\lim_{n\to\infty} \frac{\log n}{n^{0.001}} = 0$ then, for sufficiently large n we will have $\frac{\log n}{n^{0.001}} < 1$.

(e) Just note that $(1+\sin \pi n/2)/2$ cycles, with one period equal to $\{1/2, 1, 1/2, 0\}$. Thus, for all $n = 4k+1$ we have $(1 + \sin \pi n/2)/2 = 1$ and for all $n = 4k + 3$ we have $(1 + \sin \pi n/2)/2 = 0$. Thus for any fixed constant $c > 0$ for all $n = 4k + 1$ eventually $n^{(1+\sin \pi n/2)/2} = n > c\sqrt{n}$, and for all $n = 4k + 3$ we have $n^{(1+\sin \pi n/2)/2} = n^0 = 1$ and so $n^{(1+\sin \pi n/2)/2} = 1 < c\sqrt{n}$. Thus, neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$.

Exercise 4.2. Classify the following pairs of functions by their asymptotic relation; that is, determine if $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, both (i.e. $f(n) = \Theta(g(n))$) or neither.

(a) **[K]** $f(n) = 1/n$, $g(n) = \sin(1/n)$.

(b) **[K]** $f(n) = \log(n!)$, $g(n) = \log(n^n)$. What does this say about the growth rate between $f_1(n) = n!$ and $f_2(n) = n^n$?

(c) **[H]** $f(n) = n^{\log n}$, $g(n) = (\log n)^n$.

(d) **[H]** $f(n) = (-1)^n$, $g(n) = \tan(n)$.

(e) **[H]** $f(n) = n^{5/2}$, $g(n) = \left[\log\left(\sum_{k=0}^{\infty} \frac{4^{k+n} n^k}{k!}\right)\right]^2$.

> **Hint** — *Consider the Taylor series expansion of $h(x) = e^x$.*

Solution.

(a) Recall that $\lim_{u\to 0} \frac{\sin u}{u} = 1$. Setting $u \mapsto 1/n$, we have that, as $u \to 0$, $n \to \infty$. Thus, we have

$$\lim_{n\to\infty} \frac{\sin(1/n)}{1/n} = \lim_{u\to 0} \frac{\sin u}{u} = 1.$$

This is enough to show that $f(n) = \Theta(g(n))$ (you may like to check this!).

(b) We see that

$$\begin{aligned}\log(n!) &= \log(n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1) \\ &= \log(n) + \log(n-1) + \cdots + \log(2) + \log(1) \\ &< \log(n) + \log(n) + \cdots + \log(n) + \log(n) \\ &= n \cdot \log(n) = \log(n^n).\end{aligned}$$

Note that this holds for all $n \in \mathbb{N}_{\geq 1}$. Thus, $f(n) = O(g(n))$. On the other hand, we have that, for sufficiently large n, $n! > (n/2)^{n/2}$ and so,

$$\log(n!) > \log\left[\left(\frac{n}{2}\right)^{n/2}\right] = \frac{n}{2}\log(n/2) = \Theta(\log(n^n)).$$

In other words, $f(n) = \Omega(g(n))$, which shows that $f(n) = \Theta(g(n))$. However, it can be shown that $n! = O(n^n)$ strictly. An easy way to see this is to consider the ratio $a_n = n!/n^n$.

By the ratio test, we have

$$\begin{aligned}\frac{a_{n+1}}{a_n} &= \frac{(n+1)!/(n+1)^{n+1}}{n!/n^n}\\ &= \frac{(n+1)n^n}{(n+1)^{n+1}}\\ &= \frac{n^n}{(n+1)^n}\\ &= \left(\frac{n}{n+1}\right)^n\\ &= \frac{1}{e} < 1.\end{aligned}$$

So by the ratio test, $a_n \to 0$ as $n \to \infty$ which shows that $n! = O(n^n)$. This demonstrates that log does not necessarily preserve equivalence of functions.

(c) We show that
$$n^{\log n} \ll 2^n \ll (\log n)^n,$$
where $f(n) \ll g(n)$ is denoted to mean that $f(n) = O(g(n))$.

We prove the first half of the inequality; that is, $n^{\log n} \ll 2^n$. To see this, we can rewrite both expressions as
$$n^{\log n} = e^{\log\left(n^{\log n}\right)} = e^{(\log n)^2}, \quad 2^n = e^{\log(2^n)} = e^{n \log 2}.$$

To determine the order of the growth, we compare $(\log n)^2$ and $n \log 2$. By L'Hôpital's Rule, we have that

$$\lim_{n\to\infty} \frac{(\log n)^2}{n \log 2} = \lim_{n\to\infty} \frac{2 \log n}{n \log 2}.$$

Since $\log n < n$ for all $n > 1$, the limit is precisely 0. Thus, there exist some $N > 1$ such that $(\log n)^2 < n \log 2$ for all $n > N$. In other words, we have that

$$e^{(\log n)^2} \ll e^{n \log 2} \implies n^{\log n} \ll 2^n.$$

Now, for sufficiently large N, $\log N > 2$. So, for all $n > N$, $2^n \ll (\log n)^n$. From these two results, it follows that
$$f(n) = O(g(n)).$$

(d) We observe that $g(n) = \tan(n)$ is π-periodic and passes through $g(n) = 0$ infinitely many times. Thus, there is no value of N such that, for *every* $n > N$, $g(n) > c \cdot f(n)$ or $g(n) < c \cdot f(n)$. In other words, it is neither $O(g(n))$ nor $\Omega(g(n))$.

(e) Recall that the Taylor series expansion of $h(x) = e^x$ is

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

Thus, we have that

$$\sum_{k=0}^{\infty} \frac{4^{k+n}n^k}{k!} = \sum_{k=0}^{\infty} \frac{4^n\left(4^k n^k\right)}{k!} = 4^n \sum_{k=0}^{\infty} \frac{(4n)^k}{k!} = 4^n e^{4n}.$$

Then

$$\begin{aligned} g(n) &= \left[\log\left(\sum_{k=0}^{\infty} \frac{4^{k+n}n^k}{k!}\right)\right]^2 \\ &= \left(\log\left(4^n e^{4n}\right)\right)^2 \\ &= \left(\log(4^n) + \log(e^{4n})\right)^2 \\ &= (n\log 4 + 4n)^2 \\ &= c \cdot n^2 \\ &= O(n^{5/2}). \end{aligned}$$

In other words, $f(n) = \Omega(g(n))$.

Exercise 4.3. [H] Define $f : \mathbb{N} \to \mathbb{R}$ by

$$f(n) = \begin{cases} 1 & \text{if } n \leq 2, \\ f\left(\left\lceil \frac{n}{\log_2 n} \right\rceil\right) + n & \text{if } n \geq 3. \end{cases}$$

Show that $f(n) = \Theta(n)$.

Hint — *Try showing that $f(n) = O(n)$ inductively and then $f(n) = \Omega(n)$ to conclude that $f(n) = \Theta(n)$.*

Solution. Throughout the proof, we will omit the ceiling because it is negligible in asymptotic complexity analysis. We show that $f(n) = \Theta(n)$ by first showing that $f(n) = O(n)$ and then $f(n) = \Omega(n)$. We make the following claim.

Claim 4.1 — For $n \geq 512$, $f(n) \leq 2n$.

Proof. We proceed with *strong induction.*

The base case is easy to check; we have

$$f(512) = 935 \leq 1024 = 2 \cdot 512.$$

Now assume that $f(k) \leq 2k$ for all $512 \leq k < n$. Now since $k \geq 512$, we have that $\log_2 k \geq \log_2 512 = 9$. Thus,

$$\frac{n}{\log_2 512} \leq \frac{n}{9}.$$

Then we have that

$$\begin{aligned}
f(n) &= f\left(\frac{n}{\log_2 n}\right) + n \\
&\leq f\left(\frac{n}{9}\right) + n && (f \text{ is increasing}) \\
&\leq 2 \cdot \frac{n}{9} + n && (\text{inductive hypothesis}) \\
&= n\left(\frac{2}{9} + 1\right) \\
&\leq 2n.
\end{aligned}$$

■

This shows that $f(n) = O(n)$. We now show that $f(n) = \Omega(n)$. To do this, it is enough to show that $f(n) \geq n$ for $n > 2$. Note that $n/\log_2 n \geq 1$ for $n > 2$. Hence, we have that

$$f(n) = f\left(\frac{n}{\log_2 n}\right) + n \geq f(1) + n = n + 1 > n.$$

Thus, $f(n) = \Omega(n)$ and combining this result with the previous result, we have that $f(n) = \Theta(n)$.